Infer: A Statically-typed Dialect of Scheme

Preliminary Tutorial and Documentation

Version 0.1

— Limited distribution draft —

Christopher T. Haynes[1]

Computer Science Department
Lindley Hall
Indiana University
Bloomington, IN 47405 USA

chaynes@cs.indiana.edu
[812]855-3376

March 3, 1993

[1]This work was supported in part by NSF Grants CCR 87-02117 and DCR 85-21497.
Copywrite © 1992 by Christopher T. Haynes.

Contents

1 Introduction

Infer is a statically typed language designed to be accessible to Scheme programmers. It is suitable for implementation on top of Scheme, thereby providing compatibility with Scheme programming environments. Since Infer supports many features of the ML type system, plus a number of others, Infer may also be an attractive alternative to other languages supporting ML-style polymorphism. Finally, the Infer implementation, written in Infer, is well suited as a test bed for experimentation with a wide range of modern type systems.

The reader is assumed to be familiar with Scheme [?, ?, ?] and basic type system concepts [?]. Some familiarity with ML [?] would be helpful.

Infer's design owes most to ML and Scheme, but draws on other published work and includes original elements. The treatment of polymorphic assignment is due to Tofte [?]. The typing of continuations is that of Duba, Harper, and MacQueen [?]. The record type inference technique is due to Rémy and Wand [?, ?, ?]. The error reporting mechanism is inspired by Wand [?].

It is hoped that the language features documented here will prove relatively stable, but Infer's design is not complete. The greatest omission at present is support for object oriented programming and/or modules. Features described in the implementation section are most likely to change.

Section 2 provides a tutorial introduction to Infer by way of examples. Section 3 defines the syntax of Infer and informally describes its static semantics (type system) and dynamic semantics (run time behavior). Section 4 contains notes on the current implementations, with a list of know bugs.

At present the bug list includes failure to implement some of the features described in this document. Reports of further bugs, as well as design suggestions, are welcome.

2 Tutorial

This section provides an overview of Infer and its current implementation via an annotated transcript of an interactive session. Each segment of the trascript introduces one or more new feature. You are encouraged to note what is new in each segment before reading the desription that follows. In some cases the idea will be clear from the example, so the description may be read quickly for confirmation. In other cases you cannot be expected to fully understand what is going on from the example, but it provides a concrete context in which to understand the description that follows. More formal documentation of each feature is found in sections 3 and 4.

2.1 Some basics

We begin with some features of Infer, including some features of the user interface. This includes error messages and a form that allows the type of any expression to be declared.

```
|- (+ 1 2)
: num
3
```

The expression, declaration, or directive following the `|-` prompt is entered by the user. In response to an expression, the type of the expression is displayed following a colon and then the value of the expression is printed.

```
|- newline
: (proc () unit)
|- (newline)

|-
```

Whenever an expression yields an unspecified value in Scheme, Infer assigns it type `unit`. Procedures, such as `newline`, that are invoked only for side-effect return a value of type `unit`. If the value of an expression is unprintable, as when it is a procedure or the value of type `unit`, it is not printed at top-level. If the type of an expression is `unit`, the type is not printed either.

```
|- (if (lambda (x) x) 1 2)
TYPE ERROR IN EXPRESSION:
(if (lambda (x) x) 1 2)
CONFLICTING TYPES:
bool
AND
(proc ('823) '823)
```

Most type errors are detected when Infer deduces that two types should be the same, but they are not. In such cases the two offending types are displayed, along with the expression that was being checked at the time the error was detected and sometimes additional information. Non-generic type variables, such as `'823`, are represented as quoted numbers. (Generic type variables are discussed in section ??.)

```
|- (lambda (x) x)
: (proc ('a) 'a)
```

Generic type variables (see section ??) are represented by quoted identifiers.

```
|- (the (proc (num) num)
     (lambda (n) (+ 1 n)))
: (proc (num) num)
|- (not (the bool 4))
TYPE ERROR IN EXPRESSION:
(the bool 4)
CONFLICTING TYPES:
bool
AND
num
```

A `the` expression both asserts the type of an expression and returns its value. This may be useful as documentation (the correctness of which is verified), or as an aid in locating type errors.

```
|- (the (proc (num) (proc (num) num))
     (lambda (x)
       (lambda (y)
         (+ x y))))
: (proc* (num num) num)
```

The type of a unary procedure that returns a unary procedure is always printed using the type constructor `proc*`. This makes the types of curried procedures more compact. The `proc*` type constructor can also be used in the type component of a `the` expression, but this is not required since `proc*` types are equivalent to their expanded forms for type checking purposes.

2.2 Definitions

Declarations are provided for values, types, and syntactic extensions. We look first at value declarations, or *definitions*.

```
|- (define fact
     (lambda (n)
       (if (zero? n)
           1
           (* n (fact (- n 1))))))
fact : (proc (num) num)
```

In response to a declaration, an indication of the addition to the top-level type environment is provided. For a definition, the name of the defined binding and its type are displayed. Almost all of the standard Scheme procedures are provided by Infer, usually with the expected type.

```
|- (define even
     (lambda (n)
        (if (zero? n)
            #t
            (odd (- n 1)))))
even : (proc (num) bool)
WARNING: FORWARD REFERENCE: odd
|- (define (odd n)
     (if (zero? n)
         #f
         (even (- n 1))))
odd : (proc (num) bool)
|- (even 3)
: bool
#f
```

Top-level mutual recursion is supported by allowing forward references. Since what appears to be a forward reference may be a mispelled variable name, a warning message is normally printed. (See section ??.) Infer supports Scheme's lambda-eliminating shorthand for definitions, as in the definition of odd.

```
|- (define id 1)
id : num
|- (define id 0)
id : num
|- (define id (lambda (x) x))
CONFLICTING TYPES:
num
AND
(proc ('780) '780)
DEDUCED FROM EXPRESSIONS:
(define id 1)
AND
(define id (lambda (x) x))
```

Redefinition is allowed only when the type of the new value is the same as the type of the old value (or more general, see section ??).

When a type conflict is detected, the expression being checked at the time is always printed as a possible culprit. It may well be, however, that the problem lies elsewhere. Thus Infer prints the types of all expressions that were used in deducing any part of the types that are in conflict. In the above example, the expression `(define id 1)` was used in deducing the type `num`, while the expression `(define id (lambda (x) x))` was used in deducing the type `(proc ('780) '780)`.

```
|- #(init)
|- id
: '781
ERROR: VARIABLE NOT DEFINED: id
|- (define id (lambda (x) x))
id : (proc ('a) 'a)
```

Directives, of the form `#(...)`, may be issued immediately following the top-level prompt or at the outermost level of an Infer source file. The `init` directive restores the type environment to its initial state (or to the state of the most recent definition of `init-point`, see section ??). In this example the `init` directive removes `id` from the environment, so it may be redefined.

2.3 Tuples and records

An ampersand (&) introduces both tuple expressions and tuple types, while a dollar sign ($) introduces both record expressions and types. The elements of both tuples and records are selected using a percent sign (%) form.

```
|- (define a-tuple
     (& #t #\a 5 (current-input-port)
       "string" 'symbol))
a-tuple : (& bool char num port str sym)
```

The tuple created in this example contains an element of each primitive type.

```
|- (define a-record
     ($ (a 3) (b 'c)))
a-record : ($ (a num) (b sym))
|- (% b a-record)
: sym
c
```

```
|- (% 1 a-tuple)
: char
#\a
```

Records and tuples may both contain elements of differing type. Record elements are selected via a field name and tuple elements are selected via a zero-based index.

```
|- (& '(a b c) '#(1 2 3))
: (& (list sym) (vec num))
(& (a b c) #(1 2 3))
|- '(x 5)
TYPE ERROR IN EXPRESSION:
'(x 5)
CONFLICTING TYPES:
num
AND
sym
```

Lists and vectors may be quoted as in Scheme, but all elements of a list or vector must be of the same type. We will often find it convenient to use tuples as above to represent in a compact transcript the result of typing and evaluating several expressions.

2.4 Type abbreviations

Type abbreviations, or *abbrev types,* are indicated by the => punctuation within a type declaration.

```
|- (abbrevtype rtype
     ($ (left num) (right char)))
Datatype: rtype
|- (define get-left3
     (lambda (r)
       (% left (the rtype r))))
get-left : (proc (rtype) num)
```

Type abbreviations are useful when big type expressions must be repeated or when it is helpful to name type expressions. When abbreviated types are printed, they may or may not appear in their abbreviated form.

```
|- (abbrevtype inf-num-list
     ($ (head num)
        (tail inf-num-list)))
ERROR: CIRCULAR TYPE: inf-num-list
```

Recursive type abbreviations are not allowed.

2.5 Data types

Union types are called *data types* (as in ML). They always have associated names. The `datum`, `list`, and `option` datatypes are provided (see sections **??** and **??**). All others must be introduced by a `datatype` declaration. This declaration automatically introduces a set of procedures that are used to create, recognize, and extract information from each variant of the data type. A form of `case` expression makes it convenient to dispatch on the variants of a data type. Variants come in three kinds, each of which has an associated form of `case` clause.

```
|- (type switch on off)
Datatype: switch
Enumerated values: on off
|- (define (toggle switch)
     (if (off? switch) on off))
toggle : (proc (switch) switch)
|- (on? (toggle on))
: bool
#f
```

A data type named `switch` is declared with two *enumerated variants* (known as "nullary constructors" or "constants" in ML), named `on` and `off`. Enumerated variants contain no information other than the identity of the variant, so each is associated with a unique value. For each enumerated variant, a constant is defined that is named after the variant and equal to its value.

As illustrated by the procedure `toggle`, type names do not conflict with variable names. It may be regarded as helpful, or confusing, to use the same name for both a variable and a type.

Each variant of a data type is recognized by a predicate defined by the type declaration. Variant predicates, such as on? and off?, are named by appending a question mark to the variant name.

```
|- (datatype num-tree
     (leaf num)
     (node ($ (left num-tree)
              (right num-tree))))
Datatype: num-tree
Constructors: leaf node
|- (define nt
     (node ($ (left (leaf 3))
              (right (leaf 4)))))
nt : num-tree
|- (leaf% (% left (node% nt)))
: num
3
```

Num-tree is declared to be a data type with two variants, leaf and node, providing a representation for binary trees of numbers. A leaf contains a number, while a node contains two subtrees in the left and right fields of a record. Since they contain information, these are referred to as *value variants*. Each is associated with a *constructor* that bears its name. Constructors, such as leaf and node, are procedures that return a new value, of their associated data type, that contains information they are passed. Associated with each constructor is a *projection procedures* that takes a constructed variant and returns its contents. Projection procedures, such as leaf% and node%, are named by appending a percent sign to their associated constructor name.

```
|- #(init)
|- (abbrevtype nt-rec
     ($ (left num-tree)
        (right num-tree))
Type abbreviation: nt-rec
WARNING: FORWARD TYPE REFERENCE:
  num-tree
|- (datatype num-tree
     (leaf num)
     (node nt-rec))
Datatype: num-tree
Constructors: leaf node
```

This declaration of the num-tree type is equivalent to the previous one, but uses the type abbreviation nt-rec. Forward references to as-yet-undefined types are allowed, but warning messages are normally issued, as with forward references to values. This allows mutually recursive type definitions, as in this example. At least one type in each chain of mutually recursive types must be a data type. The initialization directive was used in this example because it is impossible to redeclare a type unless the new declaration is identical to the existing one.

```
|- (define node1
     (lambda (nt1 nt2)
       (node ($ (left nt1)
                (right nt2)))))
node1 : (proc (num-tree num-tree)
          num-tree)
|- (define node->left
     (lambda (nt)
       (% left (node% nt))))
node->left : (proc (num-tree)
               num-tree)
|- (define node->right
     (lambda (nt)
       (% right (node% nt))))
node->right : (proc (num-tree)
                num-tree)
|- (leaf% (node->left
            (node1 (leaf 5) (leaf 8))))
: num
5
```

These procedures make it easier to create and manipulate number trees.

```
|- #(init)
|- (datatype num-tree
     (leaf num)
     (node $ (left num-tree)
             (right num-tree)))
Datatype: num-tree
Constructors: leaf node
|- (leaf% (node->right
            (node (leaf 5)
                  (leaf 6))))
: num
6
```

In this declaration of `num-tree` the `node` variant name is followed directly by a dollar sign. This *record variant* is equivalent to the corresponding value variant of the last declaration, except that the constructor `node` now is equivalent to `node1` (as defined above), and the procedures `node->left` and `node->right` (as above) are defined automatically by the type declaration.

Thus a record variant contains a record, but since in its syntax "the record type parenthesis are omitted," Infer generates a constructor procedure that takes the record fields and builds the record and also provides for each field a procedure that projects to the variant and then selects the field.

```
|- (define sum-tree
     (lambda (nt)
       (case nt
         (leaf n n)
         (node (left right)
           (+ (sum-tree left)
              (sum-tree right))))))
sum-tree : (proc (num-tree) num)
|- (sum-tree (node (leaf 3)
                   (leaf 4)))
: num
7
```

The `case` form may be used to dispatch on the variants of a data type. Each data type `case` clause starts with the variant name. If this name is followed by a single variable name, as in the `leaf` clause, the variable is bound to the contents of the variant in a lexical context that includes the rest of the clause. If the contents of the variant is a record, the name may be followed by a list containing the names of some or all of the record fields. The contents of the fields are bound to variables of the same name in the lexical context of the rest of the clause.

```
|- (datatype switch on off standby)
Datatype: switch
Enumerated values: on off
|- (define (toggle sw)
     (case sw
       (on () off)
       (off () on)
       (standby () standby)))
toggle : (proc (switch) switch)
```

In a `case` clause for an enumerated variant, the variant name is followed by an empty list, `()`. This is appropriate, since an enumerated variant contains no information other than the identity of the variant.

```
|- (define (switch-off? sw)
     (case sw
       (on () #f)
       (else #t)))
switch-off? : (proc (switch) bool)
|- (switch-off? standby)
: bool
#t
|- (define (say-if-on sw)
     (case sw
       (on () (write "It's on!"))))
say-if-on : (proc (switch) unit)
|- (say-if-on on)
It's on!
```

If the clauses of a data type case statement do not handle all variants of the data type, an else clause may be used. If not all variants are handled and there is no else clause, then each clause must return a value of type unit. In very simple uses, as in these two examples, an if statement using a variant predicate may be preferable to a case statement.

```
|- (define (yes-char? char)
     (case char
       ((#\y #\Y) #t)
       (else #f)))
yes-char? : (proc (char) bool)
```

The case form may also be used as in Scheme, with each clause beginning with a list of character, number, or boolean values.

2.6 Local declarations

The local declaration introduces a set of declarations that are local to another set of declarations. This is analogous to let and letrec, which introduce a sequence of definitions whose scope are local to their body. The local declaration may be used to obtain a limited form of type abstraction.

```
|- (local
     ((datatype (bag 'a)
(bag (list 'a))))
     (define add-to-bag
       (lambda (x b)
 (bag (cons x (bag% b)))))
     (define in-bag?
       (lambda (x b)
 (member? x (bag% b))))
     (define empty-bag (bag '())))
add-to-bag : (proc ('a (bag 'a))
       (bag 'a))
in-bag? : (proc ('a (bag 'a)) bool)
emtpy-bag : (bag 'a)
|- (bag% empty-bag)
ERROR: VARIABLE NOT DEFINED: bag%
```

The procedures associated with the local type bag are visible only to the definitions enclosed in the local declaration.

```
|- (local ()
     (define (even n)
       (if (zero? n)
           #t
           (odd (- n 1))))
     (define (odd n)
       (if (zero? n)
           #f
           (even (- n 1)))))
even : (proc (num) bool)
odd : (proc (num) bool)
```

The local form may also be used to obtain top-level mutual recursion of value (or type) declarations without messages warning of forward references.

2.7 Datums

The datum type is a primitive data type that allows structured data to be conveniently read and written, as in Scheme. Datums may also be introduced in a program as literals using a backquote (‘) or the datum form.

```
|- read
: (proc () datum)
|- (display ‘"Hello World!")
Hello World!
```

The procedure read returns a value of type datum and the procedures display and write print elements of type datum.

```
|- (& ‘#t ‘symbol ‘"string" ‘#\a)
: (& datum datum datum datum)
(& ‘#t ‘symbol ‘"string" ‘#\a)
|- (& ‘() ‘(a list) ‘#(a vector))
: (& datum datum datum)
(& ‘() ‘(a list) ‘#(a vector))
```

Any Scheme datum may be represented as an Infer datum. In fact, an Infer implementation based on Scheme should represent an Infer datum directly as the corresponding Scheme datum. Datums are preceded by a backquote when printed at top-level.

```
|- (& (sym 'a) (bool #f) (num 3)
      (char #\x) empty)
: (& datum datum datum datum datum)
(& `a `#f `3 `#\x '())
|- (str? "x")
: bool
#t
|- (vec% `#(1))
: (vec num)
#(1)
```

The datum type is a data type with a variant for each of the primitive types that have literal representations.

```
|- (pair? `(a 3))
: bool
#t
|- (left `(a 3))
: datum
`a
|- (eq? right pair->right)
: bool
#t
```

Non-empty lists, when represented as datums, are members of the pair variant. The procedures left and right provide easy access to the left and right (car and cdr) elements of pairs.

```
|- '(($ (a ,(& 1)) (b `c)) . ,@())
: (list ($ (a (& num)) (b datum))))
(($ (a (& 1)) (b `c)))
```

The notation for record, tuple, and datum literals is active within quoted literals, as well as unquote (,) and unquote-splicing (,@) notation.

```
|- (left `'3)
: datum
`quote
|- (sym% (left `'3))
: sym
quote
|- (sym% (left `(& 1)))
: sym
&
|- (num% `,`3)
: num
3
```

The unquote and unquote-splicing notation is active within backquoted literals, but record, tuple, and quoted literal notation is not active within backquoted literals.

```
|- (& (datum 3)
      (datum (cons 'x '()))
      (datum +))
: (& datum datum datum)
(& `3 `(x) `??)
|- (datum ($ (a 2)
             (b (datum (& 1 #f)))))
: datum
`($ (a 2) (b `(& 1 #f)))
```

The datum form may be used to convert any value into a corresponding datum. To characterize this correspondence, a bit of technical machinery is required. Unprintable values are represented by ??; they include procedures, continuations, ports, and, as we shall see in section **??**, in some cases enumerated values and constructed values. We indicate the printed representation of d by $\bar{d}$; thus if d were passed to the write procedure, $\bar{d}$ would be output. If we let v be the value of an expression e in the current environment, and v contains no unprintable values, then the expression (datum e) returns a datum d such that evaluation of (quote $\bar{d}$) in the same environment would yield v.

```
|- (datum on)
: datum
`,on
|- (datum (node (leaf 3)
                (leaf (+ 2 3))))
```

```
: datum
',(node (leaf 3 ) (leaf 5))
```

Recall that by previous type declarations on is an enumerated value of the `switch` type and `node` and `leaf` are constructors of the `num-tree` type. The representation of data type values as datums using `unquote` (`,`) is consistent with the correspondence between values and their datum representation stated above. For example, the expressions on and `(quote ,on)` have the same values.

```
|- on
: switch
,on
|- (write (datum on))
,on
```

The top-level read-eval-print loop uses the `datum` form to obtain the external representation of an expression before printing it.

2.8 Polymorphism

Values that have more than one type are said to be *polymorphic* or *generic*. Infer's treatment of polymorphism is based on the Hindley-Milner type system, as used in ML. The elements of this style of polymorphism are presented here. A working facility with polymorphism is likely, however, to require further study—perhaps using an ML text (such as [?]).

```
|- ((lambda (x) x) #t)
: bool
#t
|- ((lambda (x) x) 3)
: num
3
```

In the first application the identity function `(lambda (x) x)` is used as if it had type

```
(proc (bool) bool)
```

while in the second it is "used at" type

```
(proc (num) num)
```

In general, the identity function may be regarded as having an infinite number of types of the form `(lambda (`*t*`)` *t*`)`, where *t* is any type.

```
|- (lambda (x) x)
: (proc ('a) 'a)
```

The range of types that polymorphic values may assume are expressed by *type schemes* that contain *generic type variables,* which are named `'a`, `'b`, `'c`, ..., read "alpha," "beta," "gamma," ....

```
|- '()
: (list 'a)
()
```

The empty list is also polymorphic.

```
|- cons
: (proc ('a (list 'a)) (list 'a))
|- (& (cons 'a '()) (cons 1 '()))
: (& (list sym) (list num))
(& (a) (1))
```

Many of Infer's primitive procedures, such as cons, are polymorphic.

```
|- (define get-left
     (lambda (r)
       (% left r)))
: (proc (($ (left 'a) ? 'b)) 'a)
|- (get-left ($ (left 1) (right 2)))
: num
1
```

The `get-left` procedure takes any record that has an `r` field and returns the contents of that field. The type variable `'b` following the question mark at the end of a record type indicates that the record may contain additional fields.

```
|- (define compose
     (lambda (f g)
       (lambda (x)
         (f (g x)))))
compose : (proc ((proc ('a ? 'b) 'c)
                 (proc ('d ? 'e) 'a))
            (proc ('d) 'c))
|- (compose sqrt (lambda (n) (+ n 1)))
: (proc (num) num)
|- (compose not zero?)
: (proc (num) bool)
```

The type of the compose procedure contains five generic type variables. In the first use of compose, 'a, 'c, and 'd all assume the type num, while in the second use 'a and 'c assume the types bool, while 'd assumes the type int.

A type variable may appear following a question mark at the end of the list of parameter types of a proc type, as do 'b and 'e in the example above. This indicates that the procedure could perhaps take more or fewer arguments than indicated in the proc type.

```
|- (define polyadic
     (lambda (a b
              &opt (c a) (d (= c 3))
              &rest z)
       (& a b c d z)))
: (proc (num 'a &opt num bool &rest 'b)
    (& num 'a num bool (list 'b)))
|- (polyadic 1 2)
: (& num num num bool (list 'a))
(& 1 2 1 #f '())
|- (polyadic 1 2 5)
: (& num num num bool (list 'a))
(& 1 2 5 #f '())
|- (polyadic #t #f 5 #t 8 9 10)
: (& bool bool num bool (list num))
(& #t #f 5 #t (8 9 10))
```

Procedures may have both *optional* and *rest* arguments. Optional arguments are declared with expressions that provide default values. For a call to supply an optional argument or rest arguments, all preceding optional arguments must be supplied. These expressions are evaluated in an environment that includes the preceding arguments. If a procedure accepts rest arguments, all arguments following those associated with optional arguments are collected in a list that is bound to the rest formal parameter (z in this example). Since they form a list, all rest arguments must be of the same type.

In the type of the first call of polyadic above, the empty list associated with the rest argument has type (list 'a), while in the type of polyadic the rest argument is associated with the type variable 'c. This illustrates that the scope of generic type variable names is restricted to the type in which they are contained.

```
|- (define fadder
     (lambda (f)
       (+ (f 2) (f 6 3))))
: (proc ((proc (num &opt num ? 'a)
           num))
    num)
|- (define optadder
     (lambda (&opt (i 1) (j 1) (k 1))
       (+ i j k)))
: (proc (&opt num num num) num)
|- +
: (proc (&rest num) num)
|- (& (fadder optadder) (fadder +))
: (& num num)
(& 14 11)
```

Since f is invoked with both one and two arguments, we can infer that its second argument must be optional. It may, however, be a procedure whose first argument is also optional, or a procedure that has additional optional arguments, or even a procedure such as + that takes any number of arguments.

```
|- (let ((f (lambda (x) x)))
     (& (f 1) (f #f)))
```

```
: (& num bool)
(& 1 #f)
|- ((lambda (f) (& (f 1) (f #f)))
    (lambda (x) x))
TYPE ERROR IN EXPRESSION:
(f #f)
CONFLICTING TYPES:
num
AND
bool
IN TYPES:
(proc (num) '843)
AND
(proc (bool) '843)
DEDUCED FROM EXPRESSIONS:
(f 1)
```

Polymorphic values that are bound using `let`, `letrec`, and `define` may be used polymorphically, but `lambda` bound variables may *not* be used polymorphically. This restriction is fundamental to the Hindley-Milner type discipline. It allows types to be inferred automatically.

```
|- id
: (proc ('a) 'a)
|- (define id
     (lambda (x)
       (+ (- x 1) 1)))
CONFLICTING TYPES:
'a
AND
num
IN TYPES:
(proc ('a) 'a)
AND
(proc (num) num)
DEDUCED FROM EXPRESSIONS:
(define id (lambda (x) x))
AND
(+ x (- x 1))
(define id
  (lambda (x)
    (+ x (- x 1))))
```

A value introduced by a definition may not be redefined to have a less general type.

```
|- #(init)
|- (define id
     (lambda (x)
       (+ (- x 1) 1)))
id : (proc (num) num)
|- (define id (lambda (x) x))
|- id
: (proc (num) num)
```

It is possible to redefine a value to have a more general type, but the more general type is not recorded in the type environment.

```
|- (define f (lambda (x) (g x x)))
WARNING: FORWARD REFERENCE: g
: (proc ('11) '12)
|- g
: (proc ('11 '11) '12)
ERROR: VARIABLE NOT DEFINED: g
|- (define g (lambda (a b) 3))
: (proc ('a 'a) num)
|- f
: (proc ('a) num)
|- (f 'anything)
: num
3
```

Unresolved forward references are treated as if they are lambda-bound (hence not polymorphic). When they are subsequently defined, they are treated as if they were letrec-bound, as are all definitions. Hence, after a forward reference is resolved its type becomes polymorphic. Furthermore, if any types in the type environment have free occurrences of type variables that occurred in the type of the resolved reference before it was closed (such as the type of `f` in the last example), they are re-closed. Their types may now be more generic, since some of their type variables may no longer occur free in the types of lambda-bound variables.

2.9 Polymorphic assignment

If a value or binding is mutable (assignable), it is sometimes necessary to restrict its polymorphism. This is done using *imperative* type variables, which are indicated by a backquote (‘). Non-imperative type variables, indicated by a quote (’), are said to be *applicative.* A record label or a variable binding occurrence may be followed by a bang (!), which is required in case the record field or variable binding is mutable.

```
|- box
: (proc (‘a) (box ‘a))
|- (define boxed-id (box id))
boxed-id : (box (proc (‘2) ‘2))
```

The contents of a mutable location, such as a box, is always typed without using applicative type variables. To this end procedures and forms that create mutable locations always type those locations with imperative type variables, as in the type of box. Furthermore, when an imperative type variable is unified with a type, all the type variables in that type become imperative. This happened in the above example when the applicative type variable in an instance of the type of id was unified with the imperative type variable in the type of box, so that the type of boxed-id contains an imperative type variable, ‘2.

Th type variable ‘2 is not generic, since the expression (box id) is *expansive.* Variable references and lambda expressions are *non-expansive,* and all other expressions are expansive. When the type of a declaration expression in a let or letrec expression or a definition is closed (made generic), its imperative type variables cannot become generic if the expression is expansive.

```
|- (define make-boxed-id
     (lambda ()
       (box id)))
make-boxed-id
  : (proc () (box (proc (‘a) ‘a)))
```

In this example, the lambda expression is non-expansive, so the imperative type variable becomes generic.

```
|- (define v ’#(,(lambda (x) x)))
v : (vec (proc (‘5) ‘5))
|- (vector-set! v 0 (lambda (n) 3))
|- v
: (vec (proc (num) num))
#(??)
|- (define r ($ (x ! ’())))
r : ($ (x ! (list ’a)))
|- (%-set! x r ’(4))
|- (% a r)
(4)
|- (let ((x ! (lambda (x) x))) x)
(proc (‘4) ‘4)
|- (let ((lst ! ’()))
     (set! lst ’(a b))
     lst)
: (list sym)
(a b)
```

Besides boxes, the other kinds of mutable locations supported by Infer are vectors and, when they are introduced with a bang, record fields and variable bindings. As with boxes, the contents of these locations are always typed with imperative type variables. Since vectors are mutable, those parts of a quoted literal that contain a vector are copied each time the quote expression is evaluated.

```
|- (define x ! (lambda (x) x))
x ! (proc (‘5) ‘5)
|- (set! x (lambda (n) (+ n 1)))
|- x
! (proc (num) num)
```

A bang is used instead of a colon when indicating the addition of a mutable binding to the top-level environment or the type of a top-level expression that only refers to a mutable top-level binding. A mutable binding with a type that is not fully instantiated may be assigned a value with a less general type. A binding may only be assigned if it is mutable, though redefinition may be used to change the value of any defined binding, provided the type of the new value is at least as general as the type of the old value.

2.10 Continuations

Continuations are typed as procedures, but since they are lambda-bound, they are not polymorphic.

```
|- (call-with-current-continuation
     (lambda (k)
       (+ 7 (if #t (k 2) (k 3)))))
: num
2
```

In this example the continuation k is invoked via simple applications. Since the application (k 2) is invoked, the value 2 is returned as the result of the entire expression. The call-with-current-continuation procedure is also bound to call/cc.

```
|- (call/cc
     (lambda (k)
       (if (k 1) (k 2) 3)))
TYPE ERROR IN EXPRESSION:
(if (k 1) (k 2) 3)
CONFLICTING TYPES:
bool
AND
num
DEDUCED FROM EXPRESSIONS:
(k 2)
(if (k 1) (k 2) 3)
(k 1)
```

Here k must have type (proc (num) t), for some type t. but t cannot be both num and bool.

```
|- throw
: (proc ((proc ('a) 'b) 'a)
|- (call/cc
     (lambda (k)
       (if (throw k 1) (throw k 2) 3)))
: num
1
```

The procedure throw may be used to invoke continuations, in any context.

```
|- (let
     ((f (call/cc
   (lambda (k)
     (lambda (x)
       (throw k
         (lambda (y) x)))))))
     (f 1)
     (f #t))
TYPE ERROR IN EXPRESSION:
(f #t)
CONFLICTING TYPES:
bool
AND
num
IN TYPES:
(proc (num) unit)
AND
(proc (bool) unit)
...
|- call/cc
: (proc ((proc ((proc ('a) 'b)) 'a))
    'a)
```

The second call to the procedure f fails to type check because f is not polymorphic. This is turn is due to it's argument being associated with an imperative type variable, obtained via an instantiation of 'a in the type of call-with-current-continuation. We see from this example that if continuations were instead typed with an applicative type variable, 'a, the type system would be unsound.

2.11 Directives

Directives, all of the form `#(...)`, may only be used at top-level (after the `|-` prompt, or outermost in an Infer source file). We have already used the `init` directive, but there are several more.

```
|- (define r2 ($ (a ! 3)))
r2 : ($ (a ! num))
|- #(undo r2)
|- r2
ERROR: VARIABLE NOT DEFINED: r2
|- (define r2 3)
r2 : num
```

The `undo` directive removes all top-level environment entries added since the definition of the indicated value. In this context redefinitions are ignored. Thus if the indicated value has been redefined, all entries since its first definition are removed, and the effects of redefinitions of other variables are not undone. This directive may be used if a variable has been defined with a value of the wrong type, but earlier declarations are to be retained.

The variable `init-point` is defined at the end of Infer's standard preamble (see section `preamble`) and the `init` directive is equivalent to `#(undo init-point)` followed by `(define init-point unit)`. The effect of the `init` directive may be limited further by removing the preamble's definition of `init-point`, making some definitions of your own, and then defining `init-point`.

```
|- (define rproc (lambda () r))
: (proc () num)
|- #(forget r)
|- (define r #t)
r : bool
|- (rproc)
3
```

The `forget` directive removes the top-level environment entry of an indicated definition, but not other entries. A new definition, of any type, may then be made for the same variable. Procedures that refer to a variable binding are unaffected if that binding is removed from the top-level environment via a directive.

```
|- (define lst ! '())
lst ! (list '10)
|- (datatype color blue green)
Datatype: color
Enumerated values: blue green
|- (set! lst (list blue))
|- #(undo-type color)
|- (datatype color blue green yellow)
Datatype: color
Enumerated values: blue green yellow
|- (& blue (car lst))
: (& color color:1)
(& ,blue ??)
```

The `undo-type` directive removes all top-level environment entries back to the point at which the indicated type was declared. After a type has been removed, values of the type may still exist, as in the list of the above example. When the type of such a value is printed, it's name has a number appended to it following a colon. This avoids confusion with other types of the same name. Enumerated and constructed values of removed types become unprintable. Undoing a definition does not undo imperative type variable bindings that resulted from elaboration of the definition (such as the binding of `'10` in this example).

The directive `#(forget-type` *tyname*`)` is provided to remove a single type declaration. The directives `#(forget-syntax` *keyword*`)` and `#(undo-syntax` *keyword*`)` may be used similarly to remove an individual syntactic extension (see section **??**) or a given syntactic extension along with all declarations following its declaration.

```
|- (define f0
     (lambda ()
      (f1 (f2))))
WARNING: FORWARD REFERENCES: f1 f2
f0 : (proc () '11)
|- #(forwards)
f1 : (proc ('12) '11)
f2 : (proc () '12)
|- (define f1
     (lambda (n)
       (+ 1 n)))
f1 : (proc (num) num)
|- #(forwards)
f2 : (proc () num)
```

The forwards directive may be used to obtain a list of the forward references that are currently unresolved forward references and their types, as far as can be deduced from existing usage.

```
|- #(undo f1)
|- #(forwards)
f1 : (proc ('12) '11)
f2 : (proc () '12)
|- (define f1 (lambda (b) (if b 2 3)))
f1 : (proc (bool) num)
|- (define f2 (lambda () #t))
f2 : (proc () bool)
|- #(forget f1)
|- #(forwards)
|- f1
: 'a
ERROR: VARIABLE NOT DEFINED: f1
|- (f0)
: num
2
```

Definitions that resolve forward references may be undone or forgotten. If they are undone, it is as if they had never happened (except for possible bindings of non-generic type variables occurring free in the top-level type environment and assignments to mutable bindings or data structures in the undone environment). Forgotten definitions may still be indirectly accessible through remaining bindings or data structures. This is why f1 has type 'a at the end of this example.

2.12 Syntactic extensions

Infer incorporates the high-level syntactic extension mechanism of Hieb, Dybvig, and Bruggeman [?, ?], allowing users to define new forms of directives, declarations, and expressions via pattern directed syntactic transformations. This mechanism is not yet standard and is subject to change.

```
|- (define-syntax begin0
     (lambda (x)
       (syntax-case x ()
         ((begin0 exp0 exp1 ...)
          (syntax
            (let ((val exp0))
              exp1 ...
              val))))))
Syntax: begin0
|- (begin0 1 2)
TYPE ERROR IN EXPRESSION:
(begin 2 val)
CONFLICTING TYPES:
unit
AND
num
```

New syntactic forms are expanded before type checking.

2.13 Parser generation

The seq and alt syntactic extensions and inj-pp and datum-pp procedures defined in the standard preamble of Infer's implementation provide a convenient mechanism for defining well-typed procedures that parse concrete syntax of type datum to abstract syntax specified by a user-defined data type.

For each syntactic category of the grammar to be parsed, a parse procedure, or *parse-proc*,

Figure 1: Syntax for parser generator

```
parse-proc  ⟶  prim-parse-proc
             | (alt parse-proc⁺)
             | (seq parse-proc [* | +])
             | (seq (con var*) { seq-element* parse-proc {* | +}
                               | seq-element* &rest parse-proc
                               | seq-element⁺ })

seq-element  ⟶  parse-proc | (quote id)
```

is defined. If the abstract syntax of a syntactic category is of type *pt* (parse tree), the type of the corresponding parse procedure is

```
(proc (datum
        (proc (pt) pt)
        (proc () pt))
  pt)
```

The second and third arguments are success and failure continuations, respectively.

A primitive parse procedure, or *prim-parse-proc*, can be any Infer procedure of the above type. Parse procedures for syntactic categories that are defined via BNF production rules are conveniently defined using the `seq` and `alt` forms defined in the standard preamble of Infer's implementation. The syntax for using these forms is defined in figure **??**. * and $^+$ indicate "zero or more" and "one or more," respectively.

The `alt` form recognizes alternative syntactic choices, backtracking if necessary. The `seq` form recognizes sequences of syntactic elements represented as a list of data. The `quote` form matches a given identifier. The *n*th unquoted element of a `seq` form is associated with the *n*th *var* of a `seq` form, and the number of unquoted elements must equal the number of *var*s. A type error results if they are not equal. A `*` (`+`) indicates that a list of parse trees obtained by repeating the following element to the end of the list zero (resp. one) or more times is to be bound to the associated *var*.

The `inj-pp` and `datum-pp` procedures are used to construct parse procedures that perform simple injection and datum recognition operations, respectively. See section **??** for their definitions.

To illustrate, we present a parser for a simple language whose expressions can be parsed as datums. The grammar includes `lambda` expressions with a fixed number of arguments, applications, variables and `let` expressions. See figure **??** for the BNF syntax.

The abstract syntax types are declared as follows:

```
(datatype exp
  (var var)
  (app $ (rator exp)
         (rands (list exp)))
  (abst $ (formals (list var))
          (body (list exp)))
  (let^ $ (binds (list bind))
          (body (list exp))))

(abbrevtype var sym)

(datatype bind
  (bind $ (var var) (value exp)))
```

Figure 2: Syntax of a sample language

```
⟨exp⟩  ⟶  ⟨var⟩
        |  (⟨exp⟩ ⟨exp⟩*)
        |  (lambda (⟨var⟩*) ⟨exp⟩+)
        |  (let (⟨bind⟩*) ⟨exp⟩*)

⟨bind⟩ ⟶  (⟨var⟩ ⟨exp⟩)
```

The parse procedure for variables checks that the variable is a symbol, but not `lambda` or `let`.

```
(define var-pp
  (datum-pp
    (lambda (d)
      (case d
        (sym s
          (not (or (eq? s 'lambda)
                   (eq? s 'let))))
        (else #f)))
    sym%))
```

The parse procedure for expression may now be concisely defined as follows:

```
(define exp-pp
  (alt
    (inj-pp var-pp var)
    (seq (abst formals body)
      'lambda
      (seq var-pp *)
      exp-pp +)
    (seq (let^ binds body)
      'let
      (seq (seq (bind var exp)
                var-pp
                exp-pp)
           *)
         exp-pp +))
    (seq (app rator rands)
      exp-pp exp-pp *))
```

3 Syntax and Semantics

In this section type information is given only when it would not be understood based on a basic knowledge of polymorphic type systems. The notation used in this section is summarized in figure ??. Meta-variables are indicated by italics.

3.1 Lexical conventions

Infer adopts the lexical conventions of Scheme. Thus, for example, identifiers are case-insensitive and comments begin with a semicolon and extend to the end of the line.

3.2 Syntactic classes

See figures ?? and ?? for the context free syntax of Infer's core.

3.3 Declarations

Declarations may appear at top-level or internally at the begining of a *body*, in the manner of Scheme's internal definitions. As in Scheme, a sequence of internal definitions (without other kinds of declarations intervening) is equivalent to a single `letrec`. The scope of each top-level declaration includes all following declarations.

3.3.1 Type abbreviations

(abbrevtype *paramty ty*)

Declares *tyname* $*$.. of *paramty* to be an abbreviation for *ty* parameterized by *tyvar*, $*$.. of *paramty*. That is, whenever (*tyname* ty_1 $*$..) appears as a type, it is equivalent to a reference to *ty* with each reference to *tyvar*, $*$.. replaced by ty_1, $*$... In error messages, abbreviation types may appear

Figure 3: Notation

...	ellipsis, as in standard mathematical notation
$*..$	zero or more of the preceding, used as ... in `extend-syntax`
$+..$	similar to *.., but indicates one or more of the preceding
$\longrightarrow$, $\mid$	standard BNF production and alternation
[ ], { }	extended BNF optional and grouping brackets
var : *ty*	*var* has type *ty* in the top-level type environment
$\equiv$	syntactic equivalence

in expanded form. *tyvar*, $*..$ may not be repeated.

3.3.2 Data types

(`datatype` *paramty conbind* $*..$)

Declares *tyname* of *paramty* to be a datatype with variants named after the *con*, $+..$ of *conbind*. The *tyvar*, $*..$ of a *paramty*, which may not be repeated, parameterize the binding of *tyname*. That is, whenever (*tyname* ty_1 $*..$) appears as a type, it is equivalent to a reference to the datatype with each reference to *tyvar*, $*..$ replaced by ty_1, $*..$. A *tyvar* occurring in a *conbind* must also appear in the *tyvar* list of its type declaration.

The *con*, $*..$ of *conbin*, $*..$ may not be repeated. The order of *conbind* clauses is not significant. Predicates named "*con*?," $+..$ are defined that may be applied to any value of type tyname and return true only for values of the associated variant.

A *conbind* clause may be of one of three forms, corresponding to three forms of variants.

1. For each clause of the form that consists simply of a constructor name, *con*, an "enumerated" data value named "*con*" of type *paramty* and variant *con* is defined.

2. For each clause of the form (*con* [!] *ty*) a constructor (injection) procedure named "*con*" of type

   (`proc` (*ty*) *paramty*)

   is defined that returns a value of variant *con*, a projection procedure named "*con*%" of type (`proc` (*paramty*) *ty*) is defined that returns the value injected into a value of type *tyname*, and if the ! is present a procedure named "`set-`*con*`!`" of type

   (`proc` (*paramty ty*) `unit`)

   is defined that assigns a value of type *ty* to the variant.

3. For each clause of the form

   (`con $` (*lab* [!] *ty*) $*..$)

   a constructor procedure named "*con*" of type

   (`proc` (ty_1 ty_2 $+..$) *paramty*)

   is defined that returns a value of variant *con*, and a projection procedure named "*con*%" of type

   (`proc` (*paramty*)
     (`$` (lab_1 ty_1) (lab_2 ty_2) $+..$))

Figure 4: Context free syntax

program	*prog* ⟶ {*decl* | *exp* } +..
declaration	*decl* — see section **??**
expression	*exp* — see section **??**
constant	*constant* ⟶ *bool* | *string* | *num* | *char*
boolean	*bool* ⟶ `#t` | `#f`
string	*string* — a Scheme string
number	*num* — a Scheme number
character	*char* — a Scheme character constant
symbol	*symbol* — a Scheme identifier (symbol)
keyword	*keyword* ⟶ `=>` | `&` | `$` | `%` | `%-set!` | `begin` | `case` | `datum`
	| `define` | `define-syntax` | `else` | `if` | `lambda`
	| `let` | `let-syntax` | `letrec` | `letrec-syntax`
	| `local` | `or` | `quote` | `quasiquote` | `set!`
	| `syntax` | `syntax-case` | `the`
	| `type` | `unquote` | `unquote-splicing`
identifier	*id* — a Scheme identifier not beginning with `&`
label	*lab* ⟶ *id*
variable identifier	*varid* — an *id*, but not a *keyword* or `!`
type identifier	*tyid* — an *id*, but not `$`, `&`, `proc`, `proc*`, or a *primty*
type name	*tyname* ⟶ *tyid*
constructor	*con* — a *varid*, but not a *var*
variable	*var* — a *varid*, but not a *con*
fresh variable	*fvar* — a fresh *var* (to avoid variable capture)
natural number	*nat* ⟶ `0` | `1` | `2` | ...
type	*ty* ⟶ *paramty* | *primty* | *procty*
	| *recty* | *tupty* | *tyvar* | *vecty*
vector type	*vecty* ⟶ `(vec` *ty*`)`
record type	*recty* ⟶ `($ (`*lab* [`!`] *ty*`)` `*..` [`?` *tyvar*]`)`
tuple type	*tupty* ⟶ `(&` {[`!`] *ty*} `*..`[`?` *tyvar*]`)`
procedure type	*procty* ⟶ `(proc (`ty_1 `*..` [`&opt` *ty* +..]
	[`&rest` *ty* | `?` *tyvar*]`)`
	ty`)`
primitive type	*primty* ⟶ `bool` | `char` | `num` | `port` | `str` | `sym` | `syntax`
type variable	*tyvar* ⟶ *apptyvar* | *imptyvar*
applicative type variable	*apptyvar* ⟶ '*tyid* | '*nat*
imperative type variable	*imptyvar* ⟶ ‘*tyid* | ‘*nat*

(Continued in figure **??**.)

Figure 5: Context free syntax

(Continuation of figure ??.)

body	*body* ⟶ *decl* ∗.. *exp*
datum template	*template* — a Scheme datum template
formal parameter	*fp* ⟶ *var* [!]
value binding	*valbind* ⟶ *val* [!] *body*
parameterized type	*paramty* ⟶ (*tyname* *ty* ∗..)
constructor binding	*conbind* ⟶ *con*
	| (con [!] *ty*)
	| (con ¯$ (*lab* [!] *ty*) ∗..)
syntax pattern	*syntax-pattern* — see[?] and [?]
syntax template	*syntax-template* — see[?] and [?]
Scheme expression	*scheme-exp* — a Scheme expression

is defined that returns a record of the values projected into a value of type tyname. A procedures named "*con*->*lab*" of type (proc (*paramty*) *ty*) is also defined for each label lab_1, lab_2, +.. with associated type *ty*, which is equivalent to (lambda (x) (% *lab* (*con*% x))). Finally, for each label field of the form (*lab* ! *ty*), a procedure named "set-*con*->*lab*!" of type

```
(proc (paramty ty) unit)
```

is defined that assigns a value of type *ty* to the field.

Data types declared by a type declaration may be mutually recursive. Type abbreviations may not be directly recursive or indirectly recursive through other type abbreviations, though indirect recursion through data types is allowed.

If there is only one variant in a data type, its predicate is not defined.

See sections ?? and ??.

3.3.3 Definitions

```
(define var [!] exp)
```

Defines the binding of *var* to be the value of *exp* in the top-level environment. *Var* may not have been previously defined. The type of *exp* is closed to form a type scheme associated with *var* in the context of subsequent top-level declarations and expressions. In the context of *exp*, its type is not closed. The optional ! declares the binding to be mutable and forces all type variables in the type of the binding to be imperative, so they they cannot be instantiated generically.

Bindings introduced by definitions may be redefined by subsequent definitions, provided the type of the redefinition value is at least as general as the type of the original definition. The type of the original definition is maintained in the top-level environment. See section ??.

3.3.4 Local declarations

(`local` (*decl*$_1$ *..) *decl* *..)

Declarations *decl*$_1$ *.. are evaluated in sequence, extending in turn the environment of the local declaration and then declarations *decl* *.. are in sequence. The bindings provided by *decl* *.., but not those of *decl*$_1$ *.., are then used to extend the environment of the local declaration. Local declarations may be used to define abstract data types. Though reference to type names introduced in *decl*$_1$ *.. is not restricted, access to their associated procedures is limited to *decl* *...

3.3.5 Syntactic extension

(`define-syntax` *keyword* *exp*)

Exp must have type `(proc (syntax) syntax)`. See [?] and [?].

3.4 Bodies

The bodies of `lambda`, `let`, `letrec`, `let-syntax`, and `letrec-syntax` expressions consist of a sequence of zero or more declarations followed by one or more expressions. The declarations are evaluated in sequence, with the scope of each including those that follow as well as the body's expression, except for contiguous groups of definitions, which are equivalent to a single letrec. Finally, the body's expression is evaluated and its value returned.

3.5 Expressions

In addition to the following primitive expression forms, Infer has a number of derived expression forms defined by the syntactic transformations in section ??.

3.5.1 Variables

---

var

As in Scheme.

3.5.2 Constants

---

constant

As in Scheme. Has type `bool`, `string`, `num`, or `char`, as appropriate.

3.5.3 Record construction

(`$` (*lab* [!] *exp*) *..)

The order of the fields is not significant, but the labels must be distinct. Returns a record *r* such that (`$` *lab* *r*) is the value of the *exp* corresponding to lab. A ! indicates that a field is mutable.

3.5.4 Tuple construction

(`&` {[!] *exp*} *..)

Returns a tuple whose elements are the values of *exp*, *... The "length" of a tuple is the number of its elements. A ! indicates that the field following it is mutable.

3.5.5 Record and tuple selection

(`%` *lab* *exp*)
(`%` *nat* *exp*)

In the first form *exp* must evaluate to a record with a field named *lab* and the contents of that field is returned. In the second form *exp* must evaluate to a tuple and the value of the field indexed by *nat* is returned.

3.5.6 Record and tuple assignment

```
(%-set! lab exp1 exp2)
(%-set! nat exp1 exp2)
```

In the first form *exp* must evaluate to a record with a field named *lab* and the contents of that field is assigned the value of exp_2. In the second form *exp* must evaluate to a tuple and the value of the field indexed by *nat* is assigned the value of exp_2. The assigned field must be mutable. Both forms have type `unit`.

3.5.7 Data Type Case

```
(case exp
  (con { var
       | (var1 *..) } exp1 +..) *..
  [(else exp2 +..)])
```

The *con* $*$.. must be distinct and all name constructors of the same data type, which must be the type of *exp*. The case clause must also be within the scope of the declaration of this type.

If the value, v, of *exp* is associated with one of the given constructors, that constructor's case clause is entered. Otherwise, if there is an else clause, exp_2 $*$.. are evaluated in sequence and the value of the last expression is returned, and if there is no else clause no action is taken.

Each of the last expressions in each clause must have the same type, t, which is also the type of the case expression. If *con* $*$.. include all of the constructors of their data type, an else clause is not permitted. Otherwise, if there is no else clause, t must be `unit`.

When the clause matching the variant of *exp* is found, further evaluation depends on which sort of syntax form follows the constructor of the clause.

1. If it is *var*, then *var* is bound to the result of projecting the value of *exp* to the variant type of *con*, and then exp_1 $+$.. are evaluated in the scope of *var*.
2. If it is `()`, then exp_1 $+$.. are evaluated. This is most often used when *con* is an enumerated variant.
3. If it is `(`var_1 $+$..`)`, then the *con* variant must contain a record with fields named by var_1 $+$... The variables var_1 $+$.. are bound to the values of their respective fields, and then exp_1 $+$.. are evaluated in the scope of these bindings. The order of var_1 $+$.. is not significant, and they need not name all fields of the variant record.

For each clause the value of the last exp_1 $+$.. expression is returned. See section **??**.

Scheme-style case expressions that dispatch on lists of primitive values are also supported. See sections **??** and **??**.

3.5.8 Datum creation

```
(datum exp)
```

Evaluates *exp* and returns a datum corresponding to the external representation of its value. Procedures, ports, and any other unprintable values are represented by the datum `??`.

3.5.9 Value quotation

```
(quote template)
```

The type of the `quote` form is inferred from the contents of the datum. The `unquote`, `unquote-splicing`, `quasiquote`, `&`, and `$` are all "active" within a quoted datum.

3.5.10 Datum quotation

`(quasiquote` *template*`)`

Has type `datum`. Unquote subexpressions (following a comma) must have type `datum` and unquote-splicing subexpressions must have type `(list datum)`.

3.5.11 Procedure call

`(`*exp* $+..$`)`

Procedure call, as in Scheme.

3.5.12 Begin blocks

`(begin` *exp* $*..$ exp_0`)`

Sequencing, as in Scheme. *exp*, $*..$ must have type `unit`. Has the type of exp_0.

3.5.13 Conditional evaluation

`(if` exp_0 exp_1 exp_2`)`

Conditional expression, as in Scheme. exp_0 must have type `bool` and exp_1 must have the same type as exp_2, which is the type of the `if` expression.

3.5.14 Procedural abstraction

```
(lambda (fp *.. [&opt (valbind) *..]
                [&rest fp])
  body)
```

The sequence of formal declarations, *fp*, declare the required arguments. Exercising the ! option in an *fp* declares the binding to be mutable and forces type variables in the type of the binding to be imperative, so that they cannot be instantiated generically.

Optional argument declarations follow `&opt`. The *valbind* bodies are evaluated in sequence, each in the scope of all preceding required and optional arguments. A "rest" formal paramter may be declared following `&rest`, in which case the procedure will accept any number of arguments greater or equal to the number of required arguments.

After any optional arguments have been matched, the remaining arguments are formed into a list that is bound to the rest formal parameter. All rest arguments must be of the same type.

3.5.15 Value binding

```
(let ((valbind) *..) body)
(letrec ((valbind) *..) body)
```

As in Scheme. The binding types are closed to form type schemes before checking the body. In the case of `letrec` the binding types are used to type the bindings themselves, but the binding types are not closed when the bindings themselves are checked.

3.5.16 Assignment

`(set!` *var* *exp*`)`

Variable assignment, as in Scheme. Has type `unit`. *var* must have a mutable binding.

3.5.17 Type assertion

`(the` *ty* *exp*`)`

Asserts that the type of *exp* must be *ty*, causing a type-checking error if this is not the case, and returns the value of *exp*.

3.5.18 Syntax generation

```
(syntax syntax-template)
(syntax-case exp (id *..)
  (syntax-pattern [fender-exp] output-exp)
  *..)
```

Used to define syntactic extensions in conjunction with `define-syntax`, `let-syntax`, or `letrec-syntax`. See [?] and [?]. Vector elements are treated like other subelements in a *pattern* or *template*. Both `syntax` and `syntax-case` forms have type `syntax`. *Exp* and each *output-exp* must have type `syntax`. Each *fender-exp*, if present, must have type `bool`.

3.5.19 Syntax binding

```
(let-syntax ((keyword exp) *..)
  body)
(letrec-syntax ((keyword exp) *..)
  body)
```

The *exp* must have type `(proc (syntax) syntax)`. See [?] and [?].

3.6 Derived forms

Patterns on the left- and right-hand sides of syntactic equivalences may contain optional, alternative, and ellipsis, as well as pattern variables (indicated by italics). When optional, alternative, and ellipsis appear in the right-hand side of syntactic equivalences, they refer to the corresponding optional, alternative, and ellipsis in the left-hand side, where the correspondence is determined by shared pattern variables. For example,

```
(a [x | (b y *..)])
≡ (c [(d x) | y *..])
```

abbreviates the rules:

1. `(a) ≡ (c)`
2. `(a x) ≡ (c (d x))`, and
3. `(a (b y *..)) ≡ (c y *..)`

3.6.1 *decl*

```
(begin decl *..) ≡ (local () decl *..)

(define (fp x *..) exp +..)
≡ (define fp (lambda (x *..) exp +..))

(abbrevtype tyname ty)
≡ (abbrevtype (tyname) ty)

(datatype tyname conbind *..)
≡ (datatype (tyname) conbind *..)
```

3.6.2 *body*

```
decl *.. exp +..
≡ decl *.. (begin exp +..)
```

3.6.3 *exp*

```
'template ≡ (quote template)

`template ≡ (quasiquote template)

(begin) ≡ (begin unit)

(if exp0 exp1) ≡ (if exp0 exp1 unit)

(let var ((fp exp) *..) exp0 +..)
≡ (letrec ((var (lambda (fp *..)
                  exp0 +..)))
    (var exp *..))

(case exp
  ((x *..) exp1 +..) *..
  [(else exp2 *..)])
≡ (let ((fvar exp))
(cond ((or (eqv? fvar 'x) *..)
       exp₁ +..) *..
      [(else exp₂ +..)]))
```

3.6.4 *ty*

tyname ≡ (*tyname*)

(proc* (ty_1) ty_2) ≡ (proc (ty_1) ty_2)

(proc* (ty_1 ty_2 +..) ty_3)
≡ (proc (ty_1) (proc* (ty_2 +..) *ty3*))

3.7 Standard types

A few data types and a type abbreviation must be primitive, for they are used by standard procedures. All of the procedures that would be created by the type declarations in this section are provided as standard procedures. The `datum` and `list` types are implemented specially in an Infer implementation based on Scheme.

Values of type `datum` have the same external and literal representation as in Scheme, but (unless they are unprintable) they appear (with one exception noted below) to the Infer programmer as if they were elements of a type defined as follows:

```
(datatype datum
  empty
  (bool bool)
  (char char)
  (num num)
  (str str)
  (sym sym)
  (vec (vec datum))
  (pair $ (left ! datum)
          (right ! datum)))
```

The empty list is represented by `empty`. Using the `datum` form, it is possible for unprintable values to be of type datum. Unprintable values are either opaque objects, such as procedures and ports, or values for which a ground type (a type without type variables) cannot be inferred. They are represented by the datum enumerated variant `??`. Though there is an enumerated value `??`, and a corresponding predicate `???`, there is no corresponding projection procedure (`??%`). See section **??**.

```
(datatype unit unit)
```

`Unit` is the type of statements (expressions evaluated for effect only).

```
(datatype (list 'a)
  nil
  (cons $ (hd 'a) (tl (list 'a))))
```

In an Infer implementation built on top of a Lisp (Scheme) implementation, the variants of the `list` type should be implemented as the empty list and cons cells.

```
(datatype (option 'a)
  absent
  (present 'a))

(abbrevtype (alist 'a 'b)
  (list (& 'a 'b)))
```

The `option` and `alist` types are used by some of Infer's primitive procedures.

3.8 Standard procedures

The standard procedures of Infer are those documented in this section plus those associated with the standard data types of section **??**.

```
datum->list
  : (proc (datum) (list datum))
list->datum
  : (proc ((list datum)) datum)
```

Procedures that coerce between datum and list of datum types. The argument of `datum->list` must be a list built of pairs.

```
member? : (proc ('a (list 'a)) bool)
memq? : (proc ('a (list 'a)) bool)
memv? : (proc ('a (list 'a)) bool)
```

Same as Scheme's `member`, `memq`, and `memv`, except returned values are boolean.

```
member : (proc ('a (list 'a))
           (list 'a))
memq : (proc ('a (list 'a)) (list 'a))
memv : (proc ('a (list 'a)) (list 'a))
```

Same as Scheme's `member`, `memq`, and `memv`, except the empty list is returned if no matching element is found in the list.

```
assoc : (proc ('a (alist 'a 'b))
          (option 'b))
assq : (proc ('a (alist 'a 'b))
         (option 'b))
assv : (proc ('a (alist 'a 'b))
         (option 'b))
```

These are similar to the Scheme procedures of the same name, but they return a value of type `(option 'b)`, rather than a boolean or a value of type `(& 'a 'b)`. The `option` and `alist` types are defined in the standard preamble.

```
identifier? : (proc (syntax) bool)
bound-identifier=?
  : (proc (syntax syntax) bool)
free-identifier=?
  : (proc (syntax syntax) bool)
implicit-identifier
  : (proc (syntax sym) syntax)
```

These procedures are used to define syntactic extensions in conjunction with `define-syntax`, `let-syntax`, or `letrec-syntax`. See [?] and [?].

The following standard Scheme procedures are provided with the indicated types. Infer supports all other procedures of Standard Scheme. See also the standard preamble (section ??). The complex number procedures are supported only if the implementation supports complex numbers.

```
* : (proc (&rest num) num)
+ : (proc (&rest num) num)
- : (proc (num &opt num) num)
/ : (proc (num &opt num) num)
< : (proc (&rest num) bool)
<= : (proc (&rest num) bool)
= : (proc (&rest num) bool)
> : (proc (&rest num) bool)
>= : (proc (&rest num) bool)
abs : (proc (num) num)
acos : (proc (num) num)
angle : (proc (num) num)
apply : (proc ((proc (? 'a) 'b) ? 'a)
          'b)
append : (proc (&rest (list 'a))
           (list 'a))
asin : (proc (num) num)
atan : (proc (num) num)
call-with-current-continuation
  : (proc ((proc ((proc ('a) 'b))
              'a))
      'a)
call-with-input-file
  : (proc (string (proc () 'a)) 'a)
call-with-output-file
  : (proc (string (proc () 'a)) 'a)
ceiling : (proc (num) num)
char->integer : (proc (char) num)
char-alphabetic? : (proc (char) bool)
char-ci<=? : (proc (char char) bool)
char-ci<? : (proc (char char) bool)
char-ci=? : (proc (char char) bool)
char-ci>=? : (proc (char char) bool)
char-ci>? : (proc (char char) bool)
char-downcase : (proc (char) char)
char-lower-case? : (proc (char) bool)
```

```
char-numeric? : (proc (char) bool)
char-upcase : (proc (char) char)
char-upper-case? : (proc (char) bool)
char-whitespace? : (proc (char) bool)
char<=? : (proc (char char) bool)
char<? : (proc (char char) bool)
char=? : (proc (char char) bool)
char>=? : (proc (char char) bool)
char>? : (proc (char char) bool)
close-input-port : (proc (port) unit)
close-output-port : (proc (port) unit)
complex? : (proc (num) bool)
cos : (proc (num) num)
current-input-port : (proc () port)
current-output-port : (proc () port)
denominator : (proc (num) num)
display
  : (proc (datum &opt port) unit)
eof-object? : (proc (datum) bool)
eq? : (proc ('a 'a) bool)
equal? : (proc ('a 'a) bool)
eqv? : (proc ('a 'a) bool)
even? : (proc (num) bool)
exact->inexact : (proc (num) num)
exact? : (proc (num) bool)
exp : (proc (num) num)
expt : (proc (num num) num)
floor : (proc (num) num)
for-each : (proc ((proc ('a) unit)
                  (list 'a))
             unit)
gcd : (proc (&rest num) num)
image-part : (proc (num) num)
inexact->exact : (proc (num) num)
inexact? : (proc (num) bool)
input-port? : (proc (port) bool)
integer->char : (proc (num) char)
integer? : (proc (num) bool)
lcm : (proc (&rest num) num)
length : (proc ((list 'a)) num)
list : (proc (&rest 'a) (list 'a))
list->vector : (proc ((list 'a))
                 (vec 'a))
list-ref  : (proc ((list 'a) num) 'a)
```

```
log : (proc (num) num)
magnitude : (proc (num) num)
make-polar : (proc (num) num)
make-rectangular : (proc (num) num)
make-string : (proc (num char) str)
make-vector : (proc (num ‘a) (vec ‘a))
map : (proc ((proc ('a) 'b) (list 'a))
        (list 'b))
max : (proc (num num) num)
min : (proc (num num) num)
modulo : (proc (num num) num)
negative? : (proc (num) bool)
newline : (proc (&opt port) unit)
not : (proc (bool) bool)
number->string : (proc (num) str)
numerator : (proc (num) num)
odd? : (proc (num) bool)
open-input-file : (proc (string) port)
open-output-file
  : (proc (string) port)
output-port? : (proc (port) bool)
peek-char : (proc (&opt port) char)
positive? : (proc (num) bool)
quotient : (proc (num num) num)
rational? : (proc (num) bool)
rationalize : (proc (num) num)
read : (proc (&opt port) datum)
read-char : (proc (&opt port) char)
real-part : (proc (num) num)
real? : (proc (num) bool)
remainder : (proc (num num) num)
reverse : (proc ((list 'a)) (list 'a))
round : (proc (num) num)
set-car! : (proc ((list 'a) 'a) unit)
set-cdr! : (proc ((list 'a) 'a) unit)
sin : (proc (num) num)
sqrt : (proc (num) num)
string : (proc (&rest char) str)
string->number : (proc (str) num)
string->symbol : (proc (str) sym)
string-append : (proc (&rest str) str)
string-ci<=? : (proc (str str) bool)
string-ci<? : (proc (str str) bool)
string-ci=? : (proc (str str) bool)
```

```
string-ci>=? : (proc (str str) bool)
string-ci>? : (proc (str str) bool)
string-length : (proc (str) num)
string-ref : (proc (str num) char)
string-set!
  : (proc (str num char) unit)
string<=? : (proc (str str) bool)
string<? : (proc (str str) bool)
string=? : (proc (str str) bool)
string>=? : (proc (str str) bool)
string>? : (proc (str str) bool)
substring : (proc (str num num) str)
symbol->string : (proc (sym) str)
tan : (proc (num) num)
truncate : (proc (num) num)
vector : (proc (‘a) (vec ‘a))
vector-length : (proc (vec) num)
vector-ref : (proc ((vec ’a) num) ’a)
vector-set!
  : (proc ((vec ’a) num ’a) unit)
write : (proc (datum &opt port) unit)
write-char
  : (proc (char &opt port) unit)
zero? : (proc (num) bool)
```

3.9 Standard preamble

The following declarations are loaded automatically when Infer is initialized. They may be undone.

```
(define call/cc
  call-with-current-continuation)

(define left pair->left)
(define right pair->right)

(define boolean? bool?)
(define number? num?)
(define null? nil?)
(define string? str?)
(define symbol? sym?)

(define car cons->hd)
(define cdr cons->tl)
(define caar
  (lambda (x)
    (car (car x))))
...
(define cddddr
  (lambda (x)
    (cdr (cdr (cdr (cdr x))))))

(define-syntax lambda*
  (syntax-rules ()
    ((lambda* (fs) exp ...)
     (lambda (fs) exp ...))
    ((lambda* (fs1 fs2 ...) exp ...)
     (lambda (fs1)
       (lambda* (fs2 ...) exp ...)))))

(define-syntax let*
  (syntax-rules ()
    ((let* ((fs exp)) body ...)
     (let ((fs exp)) body ...))
    ((let* ((fs1 exp1) (fs2 exp2) ...)
       body ...)
     (let ((fs1 exp1))
       (let* ((fs2 exp2) ...)
         body ...)))))

(define-syntax and
  (syntax-rules ()
    ((and) #t)
    ((and test) test)
    ((and test1 test2 ...)
     (let ((x test1))
       (if x (and test2 ...) x)))))

(define-syntax or
  (syntax-rules ()
    ((or) \#f)
    ((or test) test)
    ((or test1 test2 ...)
     (let ((x test1))
       (if x x (or test2 ...))))))

(define-syntax cond
  (syntax-rules ( else)
    ((cond) unit)
    ((cond ((else exp))) exp)
    ((cond ((test exp ...)
            clause ...))
     (if test
         (begin exp ...)
         (cond clause ...)))))

(define-syntax with
  (syntax-rules ()
    ((with () exp1 exp2 ...)
     (begin exp2 ...))
    ((with (lab1 lab2 ...)
       exp1 exp2 ...)
     (let ((var exp1))
       (let ((lab1 ($ lab1 var))
             (lab2 ($ lab2 var))
             ...)
         exp2 ...)))))

(define-syntax do
  (syntax-rules ()
    ((do ((var init exp) ...)
         (test exp1 ...)
         command ...)
     (letrec
       ((loop
          (lambda (var ...)
            (if test
                (begin exp1 ...)
                (begin
                  command ...
                  (loop exp ...))))))
       (loop init ...)))))

(define-syntax init
  (syntax-rules ()
    (#(init)
     #(begin
       #(undo init-point)
       #(define init-point unit)))))

(define init-point unit)
```

Figure 6: Additional syntax

expressions	*exp* $\longrightarrow$ {*dir*
directive	*dir* — see section ??
keyword	*keyword* $\longrightarrow$ `the-scheme`
	| `trace-type`
type	*ty* $\longrightarrow$ *boxty*
box type	*boxty* $\longrightarrow$ `(box` *ty*`)`

4 Implementation

This section describes the current implementation of Infer, including additions to the Infer language that are of a temporary or experimental nature and programming environment features. The implementation, currently runs under Chez Scheme (available from Cadence Research Systems. 620 Park Ridge Road, Bloomington, IN 47401) and MacScheme (available from Lightship Software, P.O. Box 1636, Beaverton, OR 97075).

GNU Emacs support is available for the Chez implementation in the form of an `infer.el` file.

4.1 Additional syntax and semantics

Additions to the syntax of declarations and expressions and the standard preamble are described in this section. Subsection 4.1.*n* extends subsection 3.*n*. Section ?? describes the syntax and semantics of directives.

4.1.2 Syntactic classes

See figure ??.

4.1.5 Expressions

`(the-scheme` *ty scheme-exp*`)`

Scheme-exp is executed as a Scheme expression and its value is returned. It is assumed the value has type *ty*. This is a hole in the type system, so it must be used with great caution. Free variables in *scheme-exp* are captured by local Infer bindings whose scope includes the `the-scheme` expression, while free variables that are not captured in this way are associated with top-level Scheme (not Infer) bindings.

`(trace-type` *id exp*`)`

Prints "`checking` *id*" just before beginning to typecheck *exp* and prints "`type` *id* `is` *ty*," where *ty* is the type of *exp*, when *exp* has been checked. There may be as yet uninstantiated, but non-generic, type-variables in *ty*. Equivalent to *exp* at runtime.

4.1.6 Derived forms

declarations

`#(begin` x `*..)` $\equiv$ x `*..`

`#(set` x`)` $\equiv$ `#(set` x `#t)`

`#(unset` x`)` $\equiv$ `#(set` x `#f)`

4.1.7 Standard types

The `datum` type declaration is extended with the clause `(dbox` *box*`)`.

4.1.8 Standard procedures

```
current-input-port
  : (proc (&opt port) port)
current-output-port
  : (proc (&opt port) port)
```

When an argument is given, the argument becomes the standard input port and standard ouput port, respectively. Returns the standard input port and standard output port, respectively.

`exit : (proc () 'a)`

Returns to Scheme. `(infer)` may then be used to return to Infer with the top-level context of the last exit.

```
expand : (proc (datum) datum)
expand-once : (proc (datum) datum)
```

Takes a datum representing an Infer expression and returns a datum representing the expression resulting from full expansion or one step of expansion of syntactic extensions within the given expression. Useful for debugging syntactic extensions.

`gensym : (proc () sym)`

Returns a new uninterned symbol.

`quit : (proc () 'a)`

Return to the operating system.

`reset : (proc () 'a)`

Returns to the outer top-level loop of Infer.

Complex numbers are not supported.

4.1.9 Standard preamble

Infer behaves as if the following code were automatically loaded. These declarations may be undone or forgotten.

```
(define-syntax with-syntax
  (lambda (x)
    (syntax-case x ()
      ((with-syntax ((p e0) ...)
         e1 e2 ...)
       (syntax
         (syntax-case (list e0 ...) ()
           ((p ...) (begin e1 e2 ...))
           )))))

(define-syntax transcript-on
  (lambda (x)
    (syntax-case x ()
      ((transcript-on file)
       (syntax
         #(set transcript-file file)
 )))))

(define-syntax transcript-off
  (lambda (x)
    (syntax-case x ()
      ((transcript-off)
       (syntax
         #(set transcript-file #f))))))

(define-syntax :
  (lambda (x)
    (syntax-case x ()
      ((: exp)
       (syntax exp))
      ((: exp1 exp2 exp3 ...)
       (syntax
         (: (exp1 exp2) exp3 ...))))))

(define-syntax when
  (lambda (x)
    (syntax-case x ()
      ((when test exp ...)
       (syntax
         (if test (begin exp ...)))))))

(define-syntax unless
  (lambda (x)
    (syntax-case x ()
      ((unless test exp ...)
       (syntax
         (if (not test) ))
         (begin exp ...)))))

(define-syntax fluid-let
  (lambda (x)
    (syntax-case x ()
      ((fluid-let ((var exp) ...)
         body ...)
       (syntax
         (let ((fvar var) ...)
           (dynamic-wind
             (lambda ()
               (set! var exp) ...)
             (lambda ()
               body ...)
             (lambda ()
               (set! var fvar) ...)
     )))))))

(define-syntax delay
  (lambda (x)
    (syntax-case x ()
      ((delay exp)
       (syntax
         (let ((val ! absent))
           (lambda ()
             (case val
               (absent ()
                 (set! val
                   (present exp))
                 exp)
               (present exp exp)
       ))))))))

(define (force future)
  (future))
```

```
(define (fprintf port str &rest datums)
  (let ((len (string-length str)))
    (let f ((i 0) (ls datums))
      (unless (= i len)
        (let ((ch (string-ref str i)))
          (cond
            ((and (char=? ch #\~)
                  (< (+ i 1) len))
             (case (string-ref str
                     (+ i 1))
               ((#\s)
                (write (car ls) port)
                (f (+ i 2) (cdr ls)))
               ((#\a)
                (display (car ls) port)
                (f (+ i 2) (cdr ls)))
               ((#\p)
                (pretty-print (car ls)
                  port)
                (f (+ i 2) (cdr ls)))
               ((#\%)
                (newline port)
                (f (+ i 2) ls))
               ((#\~)
                (write-char #\~ port)
                (f (+ i 2) ls))
               (else
                 (write-char #\~ port)
                 (f (+ i 1) ls)))))
          (else
            (write-char ch)
            (f (+ i 1) ls)))))))

(define (printf str &rest datums)
  (apply2 fprintf
    (current-output-port)
    str
    datums))

(define (error sy &opt (st "")
          &rest datums)
  (printf "~%ERROR in ~s" (sym sy))
  (unless (string=? st "")
    (display ": ")
    (apply2 printf st datums))
  (newline)
  (reset))

(define (for-each2 f lst1 lst2)
  (let loop ((x lst1) (y lst2))
    (if (null? x)
        (if (null? y)
            '()
            (error 'for-each2
  "lists of unequal length: ~p ~p"
              lst1 lst2))
        (f (car x) (car y))
        (loop (cdr x) (cdr y)))))

(define (map2 f lst1 lst2)
  (let loop ((x lst1) (y lst2))
    (if (null? x)
        (if (null? y)
            '()
            (error 'map2
  "lists of unequal length: ~p ~p"
              lst1 lst2))
        (cons
          (f (car x) (car y))
          (loop (cdr x)
                (cdr y))))))

(define (throw k v)
  (let ((x (k v)))
    (error 'throw
  "a `continuation' returned:~%~p"
      x)))

;;; Dynamic wind

;;; Ellipsis indicates omitted code
;;; A description follows the preamble

(define original-call/cc call/cc)

(define state-space ...)
```

```
(define dynamic-wind
 (lambda (prelude body postlude)
   ...))

(define call/cc ...)

(define call-with-current-continuation
  call/cc)

#(forget state-space)
#(forget original-call/cc)

;;; Datum parser generator

(define-syntax alt
  (syntax-rules ()
    ((alt pp1 pp2 pp3 ...)
     (lambda (d s f)
       (pp1 d s
            (lambda ()
              ((alt pp2 pp3 ...)
               d s f)))))
    ((alt pp1) pp1)))

(define-syntax seq
  (syntax-rules ( + *)
    ((seq pp)
     (lambda (d s f)
       (case d
         (pair (left right)
               (if (empty? right)
                   (pp left s f)
                   (f)))
         (else (f)))))
    ((seq pp *) (*syntax-loop* pp #f))
    ((seq pp +) (*syntax-loop* pp #t))
    ((seq (con var ...) pp1 pp2 ...)
     (lambda (d s f)
       ((*seq* pp1 pp2 ...)
        d
        (lambda* (var ...)
          (lambda ()
            (s (con var ...))))
        f)))))
```

```
(define-syntax *seq*
  (syntax-rules (& quote + * &rest)
    ((*seq* &rest pp)
     (lambda (d s f)
       (pp d
           (lambda (pt) ((s pt)))
           f)))
    ((*seq* pp *)
     (lambda (d s f)
       ((*syntax-loop* pp #f)
        d
        (lambda (pt) ((s pt))) f)))
    ((*seq* pp +)
     (lambda (d s f)
       ((*syntax-loop* pp #t)
        d
        (lambda (pt) ((s pt))) f)))
    ((*seq* (quote id) pp ...)
     (lambda (d s f)
       (case d
         (pair (left right)
           (if (and
                 (sym? left)
                 (eq? 'id
                      (sym% left)))
               ((*seq* pp ...)
                right s f)
               (f)))
         (else (f)))))
    ((*seq* pp1 pp2 ...)
     (lambda (d s f)
       (case d
         (pair (left right)
               (pp1 left
                    (lambda (pt)
                      ((*seq* pp2 ...)
                       right
                       (s pt)
                       f))
                    f))
         (else (f)))))
    ((*seq*) (lambda (d s f) (s)))))
```

```
(define *syntax-loop*
  (lambda (pp at-least-one)
    (lambda (d s f)
      (letrec
        ((loop
           (lambda (d acc)
             (case d
               (pair (left right)
                 (pp left
                   (lambda (pt)
                     (loop right
                       (cons pt acc)))
                   f))
               (empty ()
                 (s (reverse acc)))
               (else (f))))))
        (if (and at-least-one
                 (not (pair? d)))
            (f)
            (loop d '()))))))

(define inj-pp
  (lambda (pp inj)
    (lambda (d s f)
      (pp d
          (lambda (x) (s (inj x)))
          f))))

(define datum-pp
  (lambda (type? type%)
    (lambda (d s f)
      (if (type? d)
          (s (type% d))
          (f)))))

;;; Move init-point to this point

#(forget init-point)

(define init-point unit)
```

The dynamic-wind procedure definition is too long to be included in full above. It's type is

```
(proc ((proc () unit)
       (proc () 'a)
       (proc () unit))
  'a)
```

The arguments are referred to as the *prelude, body,* and *postlude.* Unless continuations are used to transfer control into or out of the body, the prelude, body, and postlude procedures are invoked in succession and the value returned by the body is returned as the value of the `dynamic-wind` call. In addition, if whenever control enters (leaves) the body via invocation of a first-class continuation, the prelude (resp. postlude) procedure is invoked. Control may not enter or leave the prelude or postlude procedures via first-class continuations.

4.2 Directives

Directives may be thought of as "compile-time expressions." Like declarations and expressions not contained in directives, they are evaluated immediately when they are entered at top-level and as they are encountered when loading a source file. Directives are also evaluated as they are encountered when compiling a file but not when loading an object file. Declarations and expressions not contained in directives, on the other hand, are evaluated when an object file is loaded, not when the corresponding source file is compiled.

Directives may only appear at top-level or within an `evaluate` directive.

Unless otherwise indicated, directives return `unit`.

4.2.1 Compile-time evaluation

`#(evaluate` *exp*`)`

The given expression is evaluated at the time indicated at the beginning of section ??.

4.2.2 Infer Parameters

```
#(set infer-parameter exp)
#(view infer-parameters)
```

The first form sets the indicated infer parameter to the value of *exp*. The second form returns a datum corresponding to the value of the indicated parameter. The Infer parameters, along with their permitted and default values are listed in figure ??.

Symbols of which data and programs are composed are considered equivalent even when they differ in case unless the `case-sensitive` parameter is `#t`.

The top-level prompt string is specified by the `infer-prompt` parameter.

See sections ?? and ?? for usage of the extension and `verbose-load` parameters.

The binding of a top-level Infer variable is a box that is bound to a top-level Scheme variable whose name is formed by concatinating the string indicated by the Infer parameter `top-level-prefix` with the name of the Infer variable. The Infer implementation uses the prefix "&," so to avoid corruption of the Infer system this prefix should not be used. If a Scheme implementation that hosts Infer may be corrupted by assigning to any top-level variables, the Infer implementation should be modified so the default prefix is a string that will avoid conflicts.

If the `transcript-file` parameter is a string, it names a file to which all top-level input and output is appended.

The parameters `warn-forward-refs` and `warn-forward-type-refs` parameters control the printing of warning messages when forward references are detected to values and types, respectively.

4.2.3 Loading files

```
#(load exp)
```

The expression should evaluate to a string, say *str*. If *str* has the extension indicated by the Infer parameter `infer-object-file-extension`, the effect is as if the code from which the file named by *str* was generated were loaded, except that the environment in which the source code is evaluated is Infer's initial top-level environment. Otherwise, the contents of the file named by *str* is treated as if it were read at top-level, except for top-level output. Top-level prompts and input are not printed. Type messages are not printed unless the Infer parameter `verbose-load` is `#t`. The values of top-level expressions that are not of type `unit` are printed, as well as error messages. See section ?? for the effects of a load directive in a file being compiled.

4.2.4 Compilation

```
#(infer-compile exp)
```

The expression should evaluate to a string, say *str*. The Infer source file indicated by *str* is compiled. It *str* does not end with "`.is`," this extension is added to obtain the source file name. Let *prefix* indicate the source file name less this extension. An Infer object file named "*prefix*`.io`" is output, which is suitable for loading by Infer. (Top-level references are to the top-level Infer environment and the Infer type environments is extended when the file is loaded.) Intermediate Scheme code is left in the file "*prefix*`.ss`." Compilation begins in Infer's initial top-level environment. A file named "*prefix*`.log`" is created that contains the top-level type messages and directives. Forward references to variables and

Figure 7: Infer Parameters

parameter	possible values	default value
`case-sensitive`	#t or #f	#f
`infer-object-file-extension`	any string	".io"
`infer-prompt`	any string	"|-"
`infer-source-file-extension`	any string	".is"
`log-file-extension`	any string	".log"
`proc*-print`	#t or #f	#t
`scheme-object-file-extension`	any string	".so"
`scheme-source-file-extension`	any string	".ss"
`top-level-prefix`	any string	""
`transcript-file`	any string or #f	#f
`verbose-load`	#t or #f	#f
`warn-forward-refs`	#t or #f	#t
`warn-forward-type-refs`	#t or #f	#t

types must be resolved within the file.

If a load directive is encountered while a file is being Infer-compiled and the indicated file contains Infer source code, this code is included in the compiled file in place of the load directive. If the file to be loaded, call it *f*, is named with the `.io` extension, indicating that it contains compiled code, the compilation environment is extended by loading *f* and object code is produced that will load *f* when the file currently being compiled is loaded. At the time the object file being compiled is loaded, the file indicated by *str* must be the same version of the file *f*. (This is checked by assigning each Infer object file a random serial number.)

The strings `.is`, `.io`, `.ss`, and `.log` mentioned above are the default values of Infer parameters, and thus may be changed. See section **??**.

The top-level environment in which this directive is issued is not changed. Thus a compiled file must be loaded before it can be used.

`#(scheme-compile` *exp*`)`

The expression should evaluate to a string, say *str*. The Infer source file indicated by *str* is compiled. It *string* does not end with "`.is`," this extension is added to obtain the source file name. Let *prefix* indicate the source file name less this extension. A Scheme object file named "*prefix*`.so`" is output, which is suitable for loading by Scheme. (Top-level references are to the top-level Scheme environment and the Infer type environment is not extended when the file is loaded.) Intermediate Scheme code is left in the file "*prefix*`.ss`." Compilation begins in the initial Infer environment. A file named "*str*`.log`" is created that contains the top-level type messages and directives. Forward references to variables and types must be resolved within the file.

If a load directive is encountered while a file is being Scheme-compiled, the indicated file must be an Infer source file and the effect is to include the source code in place of the load directive.

The strings `.is`, `.so`, `.ss`, and `.log` mentioned above are the default values of Infer parameters, and thus may be changed. See section **??**.

The top-level environment in which this directive is issued is not changed.

4.2.5 Undoing

```
#(undo var)
#(undo-type tyvar)
#(undo-syntax keyword)
```

Undo the bindings of top-level declarations back to the first binding of *var*, *tyvar*, or *keyword*, respectively. The effect is that of leaving the lexical scope of the undone bindings.

4.2.6 Forgetting

```
#(forget var)
#(forget-type tyvar)
#(forget-syntax keyword)
```

Remove the top-level binding of the indicated *var*, *tyvar*, or *keyword*, respectively, from the top-level environment. This allows the given variable, type, or keyword to be redefined, but does not effect any existing uses of the old variable, type, or keyword.

4.2.7 Forward references

```
#(forwards)
```

Print at top-level all variables with unresolved forward references along with their type constraints.

4.3 Top level

When an expression, *exp*, is entered at top level, where *exp* : *ty*, if *ty* is not `unit` or an unbound type variable, then "`:` *ty*" is printed followed by the value of *exp*. Values that are not of type `datum` are printed as an expression that would construct the value.

The symbol `??` is displayed when printing an opaque value (such as a procedure) or value of `unit` type. If a type name whose definition is not in the current type environment must be printed, a colon followed by a unique identification number are appended to the name.

If the `proc*-print` parameter is `#t`, the `proc*` type abbreviation is used when printing the type of a unary procedure whose range is a unary procedure.

For each variable var bound at top level with type *ty*, "*var* `:` *ty*" is printed, unless the binding is mutable, in which case "`!`" replaces "`:`"

For each `type` declaration, the name of the type is printed followed by the names of any new enumerated values and constructors..

Keyboard interrupts return control to the top level.

4.4 Bugs

1. The intermediate Scheme source file generated during compilation must not exist prior to the compilation.

2. Internal declarations other than definitions are not currently supported.

3. Circular data structures cannot be read or printed.

4. `extend-syntax` [?] is used instead of `define-syntax`.

5. If an expression is expanded by an `extend-syntax` syntactic extension, the expansion appears in error messages,

when the source might have been retained instead.

6. Internal definitions are not supported.
7. Nested quasiquotations are not expanded properly.
8. Run-time error messages sometimes betray the Scheme-based nature of the implementation.
9. Compilation in the initial Infer environment and generation of `.log` are not yet implemented.
10. This document should be more complete and indexed.
11. The interface to the Scheme debugger should be documented.
12. There should be a way for the user to have a personal preamble extension.

Acknowledgements

Hsianlin Dzeng has done most of the current Infer implementation and has contributed significantly to its design. The suggestions of many patient readers, especially Venkatesh Choppella and Hsianlin Dzeng, have improved this documentation. All the early users of Infer are especially thanked for their patience and suggestions. The syntactic extension code was derived from a Scheme implementation by Kent Dybvig and Bob Hieb.

References

[1] Cleaveland, J.C., *An Introduction to Data Types,* Addison-Wesley, 1986.

[2] Clinger, W., and Rees, J., "Revised^4 report on the algorithmic language Scheme," *Lisp Pointers 4*:3, pp. 1–55, 1991. Also available as a technical report from Indiana University, MIT, and the University of Oregon.

[3] Duba, B., Harper, R., and MacQueen, D., "Typing first-class continuations in ML," Proceedings of the Seventeenth Annual Symposium on Principles of Programming Languages, 1991, 163-173; revised version in preparation.

[4] Dybvig, R. K., *The Scheme Programming Language,* Prentice-Hall, 1987.

[5] Dybvig, R. K., "Writing hygienic macros in Scheme with syntax-case," Technical Report 356, Indiana University Computer Science Department, June 1992.

[6] Harper, R., Milner, R., and Tofte, M., *The definition of Standard ML,* MIT Press, 1990.

[7] Hieb, R., Dybvig, R. K., and Bruggeman, C., "Syntactic Abstraction in Scheme," Technical Report 355, Indiana University Computer Science Department, June 1992.

[8] IEEE Std 1178-1990, IEEE Standard for the Scheme Programming Language, Institute of Electrical and Electronics Engineers, Inc., New York, N.Y., 1991.

[9] Paulson, L., *ML for the Working Programmer,* Cambridge University Press, 1991.

[10] Rémy, D., "Type inference for records in a natural extension of ML," Technical Report 1431, Inria-Rocquencourt, May 1991.

[11] Springer, G., and Friedman, D.P., *Scheme and the Art of Programming,* MIT Press/McGraw Hill, 1989.

[12] Tofte. M., "Type inference for polymorphic references," *Information and Computation 89*:1, 1990. [[published reference?]]

[13] Wand, M., "Finding the source of type errors," Conf. Rec. of the 13th ACM Symp. on Principles of Programming Languages, 1986, 38-43.

[14] Wand, M., "Complete type inference for simple objects," Proceedings of the Second Symposium on Logic in Computer Science, 1987.

[15] Wand, M., "Corrigendum: Complete type inference for simple ojbects," Proceedings of the Third Symposium on Logic in Computer Science, 1988.